Assignment 05 - Fourier Transforms

This assignment may be viewed in the original markdown format or as a pdf (`assignment05.pdf`) which renders the equations in a readable form. If you're curious, the pdf was created using a tool called `pandoc` which "compiles" from markdown to LaTeX then from LaTeX to pdf.

Please make sure everything you submit runs using the commands you give us! Graders will not debug your code except for the most obvious and immediate fixes.

Problem 1: Visualizing Fourier Transforms of Images (12 points)

Objective: To understand the Fourier Transform in the context of images and practice using FFT libraries.

For this first problem you'll be modifying code in `fourier_display.py`.

Your input image will be in the format of a 2D array of `uint8`s. Keep in mind that Fourier Transforms yield complex numbers. You may need to use functions like `np.abs()`, `np.angle()`, `np.real()`, and `np.imag()` for managing these. You'll use NumPy's FFT functions (`np.fft.fft2()`, `np.fft.fftshift()`) to compute the Fourier Transform of the image. Note that the output of `fft2` is not in the order you might expect from class. In class, we always looked at magnitude and phase plots with the DC component (zero frequency) in the middle, but `fft2` natively returns an array with the DC component in the top left corner. This is because it makes visual sense to put the origin in the middle when plotting, but it makes sense from a code perspective to have index `[0, 0]` be zero frequency. The `np.fft.fftshift` function shifts things around and places the DC component to the middle, and `ifftshift` to undoes it. Shifting the outputs around like this is never necessary for the math to work, but it makes it easier to visualize the results.

Magnitude and Phase Plots: Implement two separate functions: `fourier_magnitude_as_image(image)` and `fourier_phase_as_image(image)`. The first should create an image which displays the logarithm of the magnitude in grayscale and the second should display the phase of the Fourier Transform using the HSV color space. Set saturation and value both to 255 and hue to the phase value. Recall that the H channel of HSV ranges from 0 to 180.

To be precise, the brightness of each pixel in the magnitude plot should be computed as $\log(\epsilon + abs(F(u, v)))$ where $F(u, v)$ is the Fourier Transform of the image and ϵ is a small constant to avoid taking the log of 0 (`eps` in the code). The range of magnitudes should then be normalized to `[0, 255]` and converted to `uint8`.

Run your functions on three test images provided: image1.jpg, image2.jpg, image3.jpg, and image4.jpg, and check that the outputs in `output_images/` match your expectations (note that the phase plots are not necessarily going to be interpretable).

Make sure your code runs from the command line and produces the correct outputs in the `output_images/` directory. Given the way argparse was set up for this file, you should be able to run all images with one command:

python fourier_display.py image1.jpg image2.jpg image3.jpg image4.jpg image5.jpg

Problem 2: Low-Pass and High-Pass Filters (15 points)

Objective: To implement and understand low-pass and high-pass filters in the frequency domain, and to understand how hybrid images work.

In this problem you'll be modifying code in `hybrid.py`. When you submit, include the images you generate in the `output_images/` directory. You must also include here the line of code you run to generate those outputs - it should look something like

```
# Command template/example:
python hybrid.py puppy.jpg kitten.jpg --low-cutoff=X --high-cutoff=Y
```

Your command:

```
# Your command:
YOUR PYTHON COMMAND HERE
```

The primary task in this problem is for you to implement the functions `high_pass` and `low_pass`. These must be implemented using only numpy operations and must not use any loops (vectorize everything!). Here's an outline of the steps you'll need to take for `low_pass`:

1. Compute the Fourier Transform of the image using `np.fft.fft2`. Note that if the image has multiple channels, i.e. shape `(h, w, 3)` for BGR images, numpy un-cleverly defaults to assuming that `(w, 3)` are the "spatial" dimensions. You can correct this by passing `axes=(0, 1)` to `fft2` to tell it that the `h` and `w` dimensions are the ones you want to transform. You do not need to call `fftshift` on the output, since we're just *manipulating* data in the Fourier domain rather than visualizing it.
2. Using the `frequency_coordinates_yx` function that has been provided to you, create a 2D array of the same shape as the image where the value of each pixel is equal to the **spatial frequency** of the sinusoid that is denoited by the corresponding pixel in the fourier-transformed image. Note that `frequency_coordinates_yx` returns the y and x frequencies separately, and the total frequency is the square root of the sum of their squares.
3. Create a **mask** of the same shape as the image where all values are between 0 and 1. Values corresponding to **low frequencies** (frequencies less than the cutoff frequency) should have values closer to 1, and values corresponding to **high frequencies** (frequencies greater than the cutoff frequency) should have values closer to 0. You are encouraged to try a few different things here. If you set a hard cutoff frequency so that all values in the mask are exactly 0 or 1, you'll get some artifacts in the output image. If you set the mask to be a Gaussian function of the spatial frequency, you'll be implementing something equivalent to a Gaussian blur (remember, the Fourier Transform of a Gaussian is a Gaussian). You can also try a linear ramp from 1 to 0, or a sigmoid function, or anything else. The main requirement is that the mask suppresses high frequencies and preserves low frequencies and is adjustable via the cutoff parameter.
4. Multiply the Fourier Transform of the image by the mask. This will suppress the high frequencies and preserve the low frequencies.
5. Take the inverse Fourier Transform of the masked result using `np.fft.ifft2`, and return just the `real` part of it. This is the output of the low-pass filter.

For the high-pass filter you'll do something very similar, but you'll invert the mask (`1 - mask`) so that it suppresses low frequencies and preserves high frequencies.

Once these are implemented, you'll need to decide on what the cutoff frequencies are. For this, you should plot, debug, print, or otherwise look at the values inside of the `frequency_coordinates_yx` array. What are their min and max values? What do you think the units are? Choose good values for the low and high cutoff frequencies. The goal is for the `output_images/puppy_kitten_hybrid.jpg` to look like a kitten up close and a puppy from far away. You'll need to choose the cutoff frequencies carefully to achieve this. **Note your chosen cutoff frequencies in the YOUR PYTHON COMMAND HERE block above.**

Problem 3: Convolution Theorem (20 points)

Objective: To implement and compare convolution in both the spatial and frequency domains. To get some practice writing python scripts "from scratch."

No code is given for this problem – you'll have to write everything yourself and put it in `convolution_theorem.py`. We will run this file and verify that it produces all the expected outputs, as detailed below.

Note that `cv.filter2D()` and `cv.matchTemplate()` implement *correlation* not *convolution*, so a helper function called `conv2D()` has been provided for you. Also, this problem will require you to combine filters together (e.g. blur then sobel), and in class we saw that this can be done by creating a new filter which is the convolution of the two filters (the associative property of convolution). However, this is a little tricky to implement correctly, so a helper function `filter_filter()` is also provided which will construct a new filter from two existing filters.

Imagine you're working for an image-processing company and I am a customer. I've asked you to implement a script for me that will do a bit of image filtering. I will tell you the kinds of inputs and outputs I expect, and the implementation is essentially up to you. (As graders in this course, we will be *running* your scripts, so make sure they conform to the specifications!)

The script I want you to write is called `convolution_theorem.py`. It should take three command-line arguments: an image, a filter, and a mode. The image should be a path to a JPG file which may be color or grayscale. The filter argument will accept a path to plain text file which contains a 2D array of numbers separated by spaces (an example is given in sobel.txt). Finally, the mode argument will be one of two strings, either `spatial` or `frequency` and will determine whether the filter is applied in the spatial or frequency domain. This is explained further below. The script should then create a new image which is the result of convolving the input image with the input filter. The output image should be saved in the `output_images/` directory with the same name as the input image, but with `_filtered` appended to the end of the filename. For example, if the input image is `image1.jpg`, the output image should be `image1_filtered.jpg`.

Example call:

python convolution_theorem.py –image=image1.jpg –filter=sobel.txt –mode=spatial

This should create a new file `output_images/image1_filtered.jpg` which is the result of convolving `image1.jpg` with the sobel filter in the spatial domain (separately for each channel in the input image). An example of this kind of output file naming is already done in `hybrid.py` from Problem 2.

Note that the `output_images` directory already contains the correct output of the above command, so you can check your implementation against it.

This script will likely be *much* easier to write if you enable assistive tools like GitHub Copilot or if you brainstorm the functions you need to implement on paper (or with ChatGPT) first. For instance, loading a text file into a 2D numpy array is a small and encapsulated problem and should probably get its own function.

Argparse provides a nice way of selecting among a preset list of options, so you can do the following for the mode:

```
parser.add_argument("--mode", choices=["spatial", "frequency"], default="spatial")
```

Convolution in the spatial domain should be familiar: just call `conv2D(image, filter)`.

Convolution in the frequency domain means: - Compute the Fourier Transform of (each channel of) the image. Beware of the `axes` argument to `fft2` as described in Problem 2. - Compute the Fourier Transform of the filter. Beware of the `s=image.shape[:2]` or "shape" argument to `fft2`, which pads zeros to the filter to make it the same size as the image. Otherwise, the Fourier transform of a 5x5 filter will just be a 5x5 array of numbers, which is not what we want. (We can pad as many zeros to the outside of a filter as we want without changing the result of the convolution) - Multiply the Fourier Transform of the image by the Fourier Transform of the filter. - Take the *real part* of the inverse Fourier Transform of the result.

You must implement the frequency domain convolution yourself using numpy operations.

In both cases (spatial and frequency), you'll need to rescale the outputs and change to `uint8` format. You can use `cv.normalize()` for this, or you can do it yourself with `np.clip()` and `np.uint8()`.

We expect and will test that `--mode=spatial` and `--mode=frequency` will produce the same output images (up to some small numerical precision errors).

Collaboration and Generative AI disclosure

Did you collaborate with anyone? Did you use any Generative AI tools? Briefly explain what you did here.

Reflection on learning objectives

This is optional to disclose, but it helps us improve the course if you can give feedback.

- what did you take away from this assignment?
- what did you spend the most time on?
- about how much time did you spend total?
- what was easier or harder than expected?
- how could class time have better prepared you for this assignment?

Submitting

Your submission should consist of a single zip file named like `firstname_lastname_assignment5.zip`. The zip file should contain all the contents of this directory, including this file (which you should have written some answers into above) and any images or other outputs generated by your code.